Overlapping communication and computation with OpenMP and MPI

Timothy H. Kaiser[a] and Scott B. Baden[b]
[a]*University of California, San Diego, San Diego Supercomputer Center, MC 0505, 9500 Gilman Drive, La Jolla, CA 92093-0505, USA*
Tel.: +1 858 534 5157; Fax: +1 858 534 5117;
E-mail: tkaiser@sdsc.edu
[b]*Computer Science and Engineering Department, University of California, San Diego, 9500 Gilman Drive, Mail Stop 0114, La Jolla, CA 92093-0114, USA*
Tel.: +1 858 534 8861; Fax: +1 858 534 7029;
E-mail: baden@cs.ucsd.edu

Machines comprised of a distributed collection of shared memory or SMP nodes are becoming common for parallel computing. OpenMP can be combined with MPI on many such machines. Motivations for combing OpenMP and MPI are discussed. While OpenMP is typically used for exploiting loop-level parallelism it can also be used to enable coarse grain parallelism, potentially leading to less overhead. We show how coarse grain OpenMP parallelism can also be used to facilitate overlapping MPI communication and computation for stencil-based grid programs such as a program performing Gauss-Seidel iteration with red-black ordering. Spatial subdivision or domain decomposition is used to assign a portion of the grid to each thread. One thread is assigned a null calculation region so it was free to perform communication.

Example calculations were run on an IBM SP using both the Kuck & Associates and IBM compilers.

Keywords: MPI, OpenMP, hybrid programming, clusters, parallel programming

1. Introduction

This paper discusses combining MPI and OpenMP on collections or clusters of shared memory processor or SMPs. For this class of machine, individual nodes are connected together with some type of network while the processors within a node share memory.

Many new large machines fall into the clusters of shared memory nodes classification. The machine used for this study was the IBM SP at the San Diego Supercomputer Center know as Blue Horizon. It contains 1,152 processors in 144 nodes. Each one of the nodes contains 8 processors that share memory. The nodes are connected with an IBM proprietary network. The machine supports a hybrid programming model with the use of OpenMP among processors within a node and MPI for passing messages between nodes. The SP is an instance from its class of machines. Other machines will have different network, processor, operating system, and compiler characteristics. The specific results given in this paper comparing run times from various tests might not apply to other machines. Most of the motivations and techniques discussed for combing OpenMP and MPI can be used on other machines.

We are interested in studying using OpenMP to facilitate overlapping communication and computation in hybrid programming for stencil-based grid programs. Several tests leading to this goal were performed using a simple 3 dimensional stencil-based program, RB3d. RB3d does Gauss-Seidel iteration with red-black ordering. It uses a combination of C++ and Fortran, with a Fortran kernel. The grid is distributed across the nodes of the machine, with MPI used for communication.

Various experiments were performed, changing the way OpenMP was applied to the program. OpenMP was first applied at the lower loop-level, within the Fortran kernel. For the next test, the directives were moved to a higher level within the program, outside of the kernel. The OpenMP is moved to a routine that calls the Fortran kernel. For this case there is a second level of domain decomposition. The OpenMP threads are assigned regions of this second level decomposition.

Moving OpenMP to a higher level facilitates overlapping communication and computation in hybrid programming. One of the OpenMP threads is assigned a small or null region in the second level of domain decomposition. This thread is then freed to perform the MPI communication. For programs with the appropriate computation to communication ratio this improves performance.

Scientific Programming 9 (2001) 73–81
ISSN 1058-9244 / $8.00 © 2001, IOS Press. All rights reserved

2. Background references

MPI and OpenMP are discussed in many texts. Pacheco [15] and Gropp [7], Lusk and Skjellum discuss MPI. Chandra [4], Menon, Dagum, Kohr, Maydan, and McDonald discuss OpenMP. The standards for the two systems are available from the OpenMP Architecture Review Board [13] at http://www.openmp.org and the Message Passing Interface Forum [14] at http://www.mpi-forum.org.

One of our motivations for studying hybrid programming is for its potential application to the KeLP (Kernel Lattice Parallelism) framework. KeLP is a class library, developed primarily by Scott Baden and Steve Fink at UCSD [1,2,6], for the implementation of portable applications in distributed computing. KeLP is based on the concept of regions. Users who write applications using KeLP supply subroutines to perform calculations on regions containing sections of a data grid. KeLP handles communication for regions on different processors. KeLP researchers are looking for ways to exploit multi tiered machines, and are interested in using threads for various region calculations. The KeLP framework may also benefit from overlapping communication and computation with OpenMP and MPI.

Hirsch [8] discusses Gauss-Seidel iteration with red-black ordering. Kumar [10] , Grama, Gupta, and Karypis also discuss a parallel implementation of the algorithm.

Regular grid stencil-based program are important in many areas of computational physics such as fluid dynamics, heat transfer, and electrodynamics. See LeVeque [11] for a discussion of the numerical solution of conservation laws on regular grids. The solution of Euler's hydrodynamics equation is discussed by Toro [16]. Leveque [12] also shows that regular grid routines can be used as the basis for adaptive grid methods with his Conservation Law Package, CLAWPACK. Balls [3] discusses an innovative method of solving Poisson's equations in parallel, using KeLP.

3. Additional motivation for combining OpenMP and MPI

The primary motivations for adopting most new programming paradigms are increased capability, efficiency, and ease of programming. Adding MPI to OpenMP programs allows users to run on larger collections of processors. Pure shared memory machines are limited in numbers of processors. Adding message passing can increase the number of processors that are available for a job. Adding OpenMP to MPI programs can also increase efficiency, and for some systems increase capability.

There can be multiple reads and writes of data to memory when data is sent using an MPI library. In the extreme, data might be copied from the sending array, to a temporary buffer, to the network, to a buffer on the other end, and finally to the receiving array. Even if shared memory is used to pass messages, the data must be copied from the send buffer to the receive buffer. With OpenMP, there is no implicit copying of data. This can lead to greater efficiency.

At the time the tests described in this paper were performed, there were additional limitations on Blue Horizon that encouraged the use of OpenMP along with MPI. While the machine has 8 processors per node the communications hardware only supported 4 MPI tasks per node when using the best communications protocol. To get access to all 8 processors, users were required to use some type of threads package such as OpenMP or Pthreads.

The implementation of MPI available on the SP, has another limitation. The memory required by each MPI task suffers from P^2 scaling, where P is the total number of MPI tasks in a job. This can be a problem for large jobs because MPI can use significant amounts of memory and there is little left for users. Accessing P processors using N threads and P/N MPI tasks cuts the memory required by N^2.

4. Combining OpenMP and MPI

The method for combining OpenMP and MPI is clearly program specific. This paper concentrates on an important class of problems, stencil-based or grid programs.

The grid program model is discussed below, followed by a description of the typical way to parallelize stencil-based grid applications on distributed memory machines. Then, two methods for creating hybrid programs by adding OpenMP are presented. The first uses loop-level OpenMP and the second uses OpenMP at a higher level to do coarse grain parallelism.

Data for stencil-based applications is stored in a 2 or 3 dimensional array or grid. Each iteration or time step of the calculation the grid values are updated. The values for a cell at iteration N+1 are a function of some set of the surrounding cells. Some example problems include a Jacobi iteration solution of Stommel's equa-

tion, or the solution of Euler's hydrodynamics equation as discussed by Toro [16].

Domain decomposition is often employed to solve these problems on distributed memory parallel machines. A portion of the grid is allocated on each processor and each processor is responsible for updating its portion of the grid. Processors do not have access to values from the entire grid. The information required to update a particular cell might be on a different processor. The required information is sent to/from the various processors using MPI messages.

Consider a simple two dimensional case with a total grid of interest of 100 $\times$ 100 and using two processors. Processor one could perform the calculation for the portion of the grid with indices 100 $\times$ (1-50) and processor two could perform the calculation for 100 $\times$ (51-100). If the values for a calculation are a function of the 4 surrounding cells, then processor one requires information from the cells 1-100 $\times$ (51). These values are sent from processor two. The values are stored in cells called ghost cells. One simple way to handle ghost cells is to actually allocate the grid bigger than the region for which the calculation is being performed. The extra storage is used for ghost cells. So in this case, processor one would allocate its grid of size 100 $\times$ (1-51) but only calculate for the region 100 $\times$ (1-50) and the region 100 $\times$ (51) contains ghost cells. Information about additional complexities of parallel grid program can be found in many references including Kaiser [9].

4.1. OpenMP loop level parallelism

Stencil-based grid programs often update values using one or more nested loops. A simple example is a Jacobi iteration with a five point stencil and a source term. Consider the subroutine do_jacobi.

```
subroutine do_jacobi(.....)
...
...
do j=j1,j2
 do i=i1,i2
  new_psi(I,j)= a1*psi(i+1,j)
   + a2*psi(i-1,j) + &
   a3*psi(i,j+1) + a4*psi(i,j-1) - &
   a5*force(i,j)
 enddo
enddo
...
...
```

Here we have old grid values stored in the array psi. These psi values are being used to calculate new grid values placed in new_psi. To "OpenMP" the code section, a parallel do directive is added.

```
!$OMP PARALLEL
!$OMP DO SCHEDULE (STATIC)
private(i)
firstprivate(a1,a2,a3,a4,a5)
do j=j1,j2
 do i=i1,i2
  new_psi(I,j)= a1*psi(i+1,j)
   + a2*psi(i-1,j) + &
   a3*psi(i,j+1) + a4*psi(i,j-1) - &
   a5*force(i,j)
 enddo
enddo
```

Pseudo code for a kernel of a Jacobi iteration program in MPI and OpenMP would be

```
do k=1,num_iterations
 call MPI_transfer_routine(...)
 call do_jacobi(...)
 psi=new_psi
enddo
```

MPI_transfer_routine passes ghost cells between nodes and do_jacobi contains the OpenMP directives as shown above.

The algorithm given above has two distinct phases. There is a communication phase followed by a computation phase. All processors must wait for the communication to complete before they perform any computation. This is true even though some of the computation is not dependent on the communication.

Stencil-based grid programs suffer from a problem: single stepping through the grid using the straight forward nested loops there is very little data reuse. Data is used once or twice and then it is flushed from cache. This leads to slower performance. It is possible to do cache blocking but doing so often destroys the clean nature of the code that is exploited by a programmer using OpenMP directives. When doing cache blocking there are additional loop levels and application of OpenMP directives becomes more problematic.

4.2. Higher level coarse grain OpenMP parallelism

Consider the algorithm discussed above. For some cells, the data for the stencil calculation is dependent on communication and for some it is independent of communication. A natural thought is to break the grid into communication dependent and independent regions. With this done, an algorithm can be written to

allow overlap of communication and computation. For this algorithm, you first start communication of data for the dependent regions. Calculation is done on the independent regions while communication continues. After communication completes, calculation can proceed for the dependent regions.

We can further decompose the regions into a collection of smaller regions. Threads can be assigned to blocks of data using OpenMP. This concept leads to the OpenMP directives being placed at a higher level in the program as shown below.

This is of interest to the KeLP developers because this concept fits well with the KeLP model, with a collection of regions assigned to a node and regions are assigned to threads. It also allows the exploration of overlapping communication and computation without rewriting the KeLP framework.

There are other advantages to moving the OpenMP directives up to a higher level in the program. For some problems moving the OpenMP directives to a higher level allows a looser level of synchronization, with a reduction of loop level synchronization cost. Also, programmers are freer to do additional optimizations at loop level, such as cache blocking that are difficult if OpenMP is applied to loops.

We define an abstract data type "region" that holds indices for a portion of the grid and an "empty" flag. For each MPI task we break the calculation grid into two sets of regions. The number of regions in each of the sets is equal to the number of threads. If we wish to overlap computation and communication one of the regions in the first set is empty. The thread that has an empty region handles communication. The second set of regions contains portions of the grid that are affected by communication. This is illustrated in Fig. 1 for a case where we have four regions in each set. The forth region in set one, is empty. Thus, the thread is assigned to this region performs communication.

After we have broken the grid into regions the algorithm proceeds as follows

```
for each timestep
 In parallel for each region in
 set 1
  If region is not empty
   Perform calculation in kernel
  Else if region is empty
   Start MPI communication
 end parallel
 finish MPI communication
 in parallel for each region in
 set 2
   perform calculation in kernel
 end parallel
end timestep
```

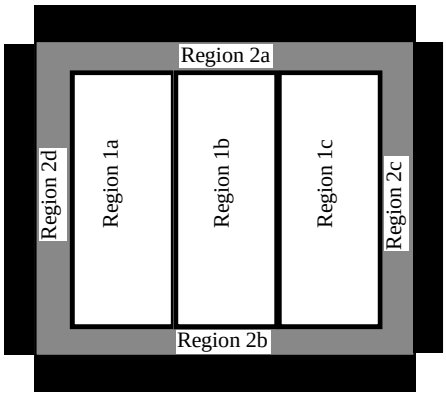


Fig. 1. Regions within a two dimensional grid. Black area is ghost cells. Regions 2a–2b are those effected by communication. Regions 1a–1c are independent of communication.

We can compare this to the algorithm shown below when OpenMP is applied in the kernel.

```
for each timestep
 Start MPI communication
 finish MPI communication
 Perform calculation in OpenMP
 enabled kernel
end timestep
```

This appears simpler, but applying OpenMP in the kernel might be difficult. Also, this does not allow overlapping of computation and communication.

When computation and communication are overlapped a processor is given up to do the communication. When is it beneficial to overlap computation and communication? Consider a single iteration step of a calculation containing both computation and communication, but no overlapping. Normalize the run time for the step to 1, and assume the communication takes time C. Computation is spread across P processors and take time $1 - C$. The work associated with the computation is $P*(1-C)$. If we use one of the threads for communication then the work must be spread across $P - 1$ processors. The run time sparing one thread for communication is

$$T = \frac{P(1-C)}{P-1}$$

There is speedup when the run time, T, is less than 1. This occurs if $C > 1/P$. If $C > P/(2P-1)$ then communication time is dominant and we get the upper limit on speedup of $T = P/(2P-1)$. If $P = 8$ then

there is speedup if $C > (1/8)$ and the speedup will peak with $C = 8/15$. That is, there is speedup using overlap if the communication takes over $1/8$ of the time for a single iteration without overlap and the maximum speedup is $S = 15/8$ when $C = 8/15$.

We clearly need asynchronous communication routines to do the overlap discussed above. MPI contains many routines to support asynchronous communication. The typical way to do this is for the sending process to "post" a send by doing a MPI_Isend. This starts the message going. The receiving process does a MPI_Irecv. After some time, both processes do an MPI_Wait to block until the communication completes. There are versions of MPI_Wait that only test to see that messages have completed and don't block. MPI has the ability to define derived data types. These can be used to efficiently send noncontiguous blocks of data such as rows of a matrix. These routines are discussed in most MPI texts. Two good ones have been given above, Pacheco [15] and Gropp [7], Lusk and Skjellum.

5. Application for testing hybrid program methodologies

The test code that was used to validate the concepts discussed in this paper was an iterative 3d Laplace's equation solver that uses the Gauss-Seidel's method with red-black ordering. RB3d. The original program has been used as a research tool in association with KeLP and is part of the KeLP distribution. The version that was used does not contain any of the KeLP system. RB3d is a simple program with a 7 point stencil. It contains a C++ driver with Fortran kernel. When the OpenMP was placed outside of the kernel, the Fortran routines were not modified from their original form. Minimum modifications were made to the rest of the program.

This program was primarily used to see if overlapping communication with computation using OpenMP is effective. IBM's C++ compiler does not support OpenMP. There is support available from the IBM C and Fortran compilers. C or Fortran routines containing OpenMP can be called from C++. Thus, the C++ routines that contained OpenMP were rewritten in C.

The program was designed with several options that are invoked based on input. Overlapping of communication and computation can be turned on or off. Blocking for cache in the Fortran kernel can also be toggled.

There is also a version that does loop-level OpenMP. This version is simply the original program with directives inserted in the Fortran kernel. The kernel is shown below. The top part is used if cache blocking is enabled. Si and sj are the variables that effect cache blocking. Note the added complexity of this block of code. Most importantly, the "best" place for the Parallel Do directive is a function of the blocking parameters.

The variable rb determines if the updated values are put into the red or black portion of the grid. That is, on one iteration of the kernel half of the grid is updated using values from the other half of the grid. This makes all updates to $u(i, j, k)$ independent of each other.

```
 if(do_blocking)then
!$OMP PARALLEL DO
firstprivate(c,c2)
loop_a:  do jj = wl1, wh1, sj
loop_b:   do ii = wl0, wh0, si
loop_c:    do k = wl2, wh2
loop_d:     do j = jj, min(jj+sj-1,
             wh1) is = ii +
             mod(j+k+ii+rb,2)
loop_e:      do i = is, min(ii+si-1,
              wh0), 2 u(i,j,k) = c*
              (u(i-1,j,k)
              +u(i+1,j,k) + &
              u(i,j-1,k)
              + u(i,j+1,k) + &
              u(i,j,k-1)
              + u(i,j,k+1) - &
              c2*b(i,j,k))
     end do loop_e
    end do loop_d
   end do loop_c
  end do loop_b
 end do loop_a
 else
!$OMP PARALLEL DO
firstprivate(c,c2)
 do k=wl2, wh2
  do j=wl1, wh1
   do i=mod(j+k+rb-1,2)+1,wh0,2
    u(i,j,k) = c * (u(i-1,j,k)
    + u(i+1,j,k) + &
     u(i,j-1,k) + u(i,j+1,k) + &
     u(i,j,k-1) + u(i,j,k+1) - &
     c2*b(i,j,k))
    enddo
   enddo
  enddo
 endif
```

The program was run using 4 MPI tasks spread on to 4 nodes. A grid of $120 \times 120 \times 240$ on each node was

Table 1
The normalized communication time is greater than 1/P so improvement is expected if communication and computation are overlapped

Single iteration run times for RB3d with normalized commincation time compared to range 1/P to P/(2P-1)					
Communication time (seconds)	Total cycle time (seconds)	P (threads)	1/P	C Normalized Communication Time	P/(2P-1)
0.0678	0.1238	4	0.25	0.55	0.57
0.0691	0.1153	5	0.20	0.60	0.56
0.0714	0.1118	6	0.17	0.64	0.55
0.0874	0.1247	7	0.14	0.70	0.54
0.0847	0.1187	8	0.13	0.71	0.53

used while calculating for 100 iterations. The number of OpenMP threads was varied from 4 to 8.

The program was first run without overlapping communication and computation for various numbers of threads. These runs were done to determine if improvements in run time could be expected using overlapping. As discussed in Section 3.2, improvement is expected if the normalized communication time is greater than $1/P$. Table 1 shows that this program has sufficient communication to improve performance with overlap. For example, with four threads and $1/P = 0.25$ we have a normalized communication time of 0.55.

The program was next run with overlapping turned on. There were actually two cases. In the first case

```
#pragma omp parallel for schedule
(dynamic,1)
 for( ig=0;ig<nthread;ig++) {
  ...
  do_kernel(region[ig]);
  ...
```

was used to start the parallel calculation for each of the regions of the grid. In the second case a parallel section was entered using a simple parallel directive

```
#pragma omp parallel
{
 ig=omp_get_thread_num();
  do_kernel(region[ig]);
```

We are allowing each thread to grab a region to work on. This second case requires that each thread execute the parallel region once and only once. This might not work on some compilers. Also in our test codes, we found no significant improvement by using a simple "parallel" directive and allowing each thread to grab a region to work on, instead of "parallel for".

The program was run using both the native IBM compilers and Kuck and Associates compilers. Also, the programs were run with cache blocking enabled and disabled. Cache blocking for grid programs is discussed by Douglas [5], Hu, Kowarschik, Rude, and Weiss.

Table 2
The run times when the OpenMP directives were placed in the Fortran kernel show poor scaling

Red Black 3d Times 4 MPI tasks Grid 120 × 120 × 240 on each node OpenMP applied in Fortran kernel			
Cache blocking	Total threads	Time IBM compiler (seconds)	Time KAI compiler (seconds)
F	4	10.979	18.218
F	5	10.108	18.193
F	6	9.132	18.232
F	7	9.021	18.181
F	8	10.536	18.071
T	4	26.815	40.185
T	5	26.046	40.436
T	6	26.210	40.207
T	7	26.915	40.390
T	8	25.855	40.180

6. Results

The best result for all tests occurred using the IBM compiler, the OpenMP applied in the Fortran kernel, and without using cache blocking. Table 2 shows the run times when the OpenMP directives were placed in the Fortran kernel. For the IBM compiler, and cache blocking turned off, the run time is reduced by 20% going from 4 to 7 threads and it starts to go back up at 8 threads. For the KAI compiler, run time is flat.

Why did the program perform relatively well when cache blocking was turned off? The IBM SP has a large l2 cache and each processor has its own cache. By using the OpenMP directives to spread the grid across all processors most of the grid stayed in cache. In this case, using OpenMP gave us the effect of cache blocking for free. Note that this would not hold for larger grids.

When using OpenMP directives with the cache blocked section of code, the performance is dependent

Table 3
Shows the speedup of using a thread to perform MPI communication with coarse grain OpenMP

Red Black 3d Times 4 MPI tasks Grid 120 × 120 × 240 on each node OpenMP Applied at higher level outside of the Fortran kernel Cache blocking enabled				
Compiler	Total threads	Time (seconds) Thread NOT used for communicaiton	Time (seconds) Thread WAS used for communicaiton	% Speedup (T1/T2-1)%
IBM	4	20.844	20.181	3.29
IBM	5	19.484	17.145	13.64
IBM	6	17.992	15.318	17.46
IBM	7	17.153	14.321	19.78
IBM	8	16.666	13.879	20.08
KAI	4	19.599	19.022	3.03
KAI	5	17.996	15.618	15.23
KAI	6	16.840	13.784	22.17
KAI	7	15.983	12.484	28.03
KAI	8	15.050	12.500	20.40

on where the directive is placed. The "cache blocking enabled" data from Table 2 was obtained with the OpenMP directive was placed before loop_a as shown in Section 3.1.

Putting the OpenMP directive above loop_a with wl1 = 1, wh1 = 120 and sj = 64 the lines

```
!$OMP PARALLEL DO
firstprivate(c,c2)
 do jj = wl1, wh1, sj
```

become at run time

```
!$OMP PARALLEL DO
firstprivate(c,c2)
 do jj = 1, 120, 64
```

So the compiler can only provide work for two threads, when jj = 1 and jj = 65. This is why the performance was flat for additional threads and why the using the "cache blocking enabled" section of code did not work well.

The directive can be placed ahead of one of the other do loops. This was tried using 7 threads and the same input parameters. Placing the directive above loop_b and loop_d produced the same results. The best results were obtained with the directive above loop_c, the run time was reduced to a little over 12 seconds, which is comparable to the non cache blocked code. This worked well because of the large range of indices for the loop and unit stride. Placing the directive in front of loop_e was disastrous, the run time jumped to 938.9 seconds.

The optimum placement is a function of the size of the grid and the cache blocking parameters. Moving the OpenMP directives to a higher level in the program removes these concerns.

A primary objective of this work is to move the OpenMP directives to a higher level and then compare using all threads for computation to using a thread to overlap communication and computation. Did reserving a thread for communication improve performance over using all threads to do computations? It did, but the best results from doing this were not quite as good as the best results discussed above.

When the directives were placed at the higher level to do coarse grain parallelism, using a thread to overlap computation and communication to reduce run time. Unlike the previous results, with high level OpenMP, turning cache blocking on improves performance. OpenMP was used to enable each thread to call its own copy of the Fortran kernel and pass it a region for which to calculate. The OpenMP directive that distributed the regions was

```
#pragma omp parallel for schedule
(dynamic,1)
 for( ig=0;ig<nthread;ig++) {
  ...
  do_kernel(region [ig]);
  ...
```

Table 3 shows run times with and without a thread reserved for doing communication. Notice that using a thread for communication improves the run time of the program even though this reduces the number of computational threads by 1. The speedup is almost 22% for the KAI compiler using 6 calculation threads and 1 communication thread, compared to using 7 threads for computation.

Another important trend in this data is that scaleability improves by reserving a thread to do communication. For the IBM compiler, doubling the number of

Table 4
Shows improved scaling by using a thread to perform MPI communication with coarse grain OpenMP

Red Black 3d Scaling 4 MPI tasks Grid 120 × 120 × 240 on each node OpenMP Applied at higher level outside of the Fortran kernel					
Compiler	Total Threads	Time (seconds) Thread NOT used for communicaiton	Scaling relative to 4 threads	Time (seconds) Thread WAS used for communicaiton	Scaling relative to 4 threads
IBM	4	20.844	1.00	20.181	1.00
IBM	5	19.484	0.86	17.145	0.94
IBM	6	17.992	0.77	15.318	0.88
IBM	7	17.153	0.69	14.321	0.81
IBM	8	16.666	0.63	13.879	0.73
KAI	4	19.599	1.00	19.022	1.00
KAI	5	17.996	0.87	15.618	0.97
KAI	6	16.840	0.78	13.784	0.92
KAI	7	15.983	0.70	12.484	0.87
KAI	8	15.050	0.65	12.500	0.76

Table 5
Program run times are faster with cache blocking enabled. Moving OpenMP to higher levels to do coarse grain parallelism facilitated cache blocking

Red Black 3d Times 4 MPI tasks Grid 120 × 120 × 240 on each node OpenMP Applied at higher level outside of the Fortran kernel Cache blocking disabled compared to cache blocking enabled					
Compiler	Total threads	Time (seconds) Thread NOT used for communicaiton	Time (seconds) Thread WAS used for communicaiton	Time (seconds) Cache blocking enabled	% Speedup Cache blocking enabled (T2/T3-1)%
IBM	4	27.316	24.824	20.181	23.01
IBM	5	28.340	24.458	17.145	42.65
IBM	6	27.370	25.484	15.318	66.37
IBM	7	27.493	25.940	14.321	81.13
IBM	8	27.754	26.938	13.879	94.09
KAI	4	26.526	24.206	19.022	27.25
KAI	5	26.572	24.211	15.618	55.02
KAI	6	26.419	24.977	13.784	81.20
KAI	7	26.831	25.644	12.484	105.41
KAI	8	27.200	26.964	12.500	115.71

processors and not using a thread for communication results in a 20% reduction in run time. Doubling the number of processors and using a thread for communication gives a 31% reduction in run time. For the KAI compiler there is a 23% reduction in run time while not using a thread for communication and a 34% reduction when a thread is used for communication.

Table 4 expresses this in terms of scalability relative to 4 threads. When a thread is not used for communication the scaling drops to about 0.64. Scaling drops less with a thread used for communication, to about 0.75. This is important for the Blue Horizon machine with 8 processors per node and can be very important on machines that have more processors per node, such as the Livermore ASCII White machine that has 16 processor per node. The general trend for new hybrid machines is to add additional shared memory processors per node.

Table 5 contains the run times of the program with and without using a thread to do communication, but with the cache blocking in the Fortran kernel disabled. Notice that the best run times are about twice of those for which cache blocking was enabled. Moving the OpenMP directives out of the Fortran kernel to a higher level enables the cache blocking optimizations to be used without interference from the OpenMP directives.

7. Conclusion

OpenMP combined with MPI can be used to program machines containing a distributed collection of shared memory nodes. Adding MPI to OpenMP programs allows users to run on larger collections of processors. Adding OpenMP to MPI programs can also increase efficiency and capability for some systems.

OpenMP with MPI were combined in a simple 3 dimensional stencil-based program using different strategies. Domain decomposition was used to distribute the

grid across the nodes of a distributed memory machine containing 8 processors per node. Loop level OpenMP was first added to the nested do loop kernel of the program. The kernel contained two different options for performing the calculation. One option used a cache blocking algorithm and the other used a serial progression through the points of the grid. OpenMP preformed well when cache blocking was not used. With cache blocking, the performance varied widely depending on the placement of the directives.

OpenMP was moved to a higher level of the program. A second level of domain decomposition was done and each thread was assigned a portion of the grid to calculate. This can potentially add to the simplicity of a program because no modifications are needed to the kernel. Users are free to use optimization strategies, such as cache blocking, in the kernels without interference from OpenMP directives. This strategy also has the advantage that a thread can be assigned a null calculation region and be reserved for performing communication. When doing coarse grain parallelizm, reserving a thread for communication significantly increased the performance and scalability of the program.

Acknowledgments

This work was funded by the National Science Foundation's National Partnership for Advanced Computational Infrastructure (NPACI) program. More information on NPACI is available at http://www.npaci.edu.

References

[1] S.B. Baden, *The KeLP Programming System,* http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.

[2] S.B. Baden and S.J. Fink, A Programming Methodology for Dual-tier Multicomputers, *IEEE Transactions on Software Engineering* **26**(3) (2000), 212–226.

[3] G. Balls, *A finite Difference Domain Decompostion Method Using Local Corrections for the Solution of Poisson's Equation,* Ph.D. Dissertation, University of California at Berkeley, 1999.

[4] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan and J. McDonald, *Parallel Programming in OpenMP,* Morgan-Kaufman, San Francisco, 2000.

[5] C. Douglas, J. Hu, M. Kowarschik, U. Rude and C. Weiss, Cache Optimization for Structured and Unstructured Grid Multigrid, *Electronic Transaction on Numerical Analysis* **10** (2000), 21–40.

[6] S.J. Fink, *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters,* Ph.D. Dissertation, University of California San Diego, 1998.

[7] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* MPI Press, Cambridge, 1994.

[8] C. Hirsch, *Numerical Computation of Internal and External Flows,* Wiley and sons, Chichester, 1988.

[9] T.H. Kaiser, *Dynamic Load Distributions for Adaptive Computations on MIMD machines using Hybrid Genetic Algorithms,* Ph.D. Dissertation, University of New Mexico, 1997.

[10] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to parallel Computing, Design and Analysis of Algorithms,* Benjammin/Cummings, Redwood City, 1994.

[11] R. LeVeque, *Numerical Methods for Conservation Laws,* Birkhauser-Verlag, Basel, 1990.

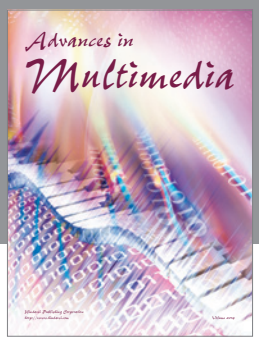
[12] R. LeVeque, *Conservaion Law Package,* http://www.amath.washington.edu/~claw.

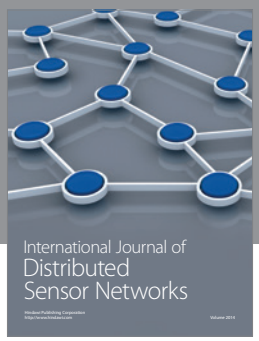
[13] Message Passing Interface Forum, Message passing Interface Standard, http://www.mpi-forum.org.

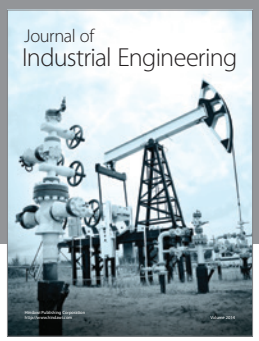
[14] OpenMP Architecture Review Board, OpenMP Standard, http://www.openmp.org.

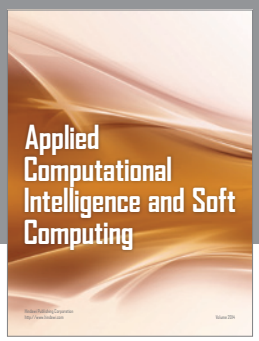
[15] P.S. Pacheco, *Parallel Programming with MPI,* Morgan-Kaufman, San Francisco, 1997.

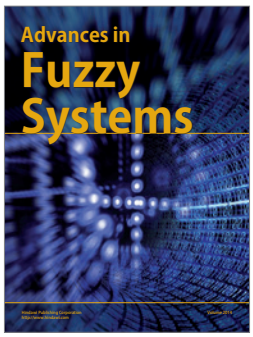
[16] E.F. Toro, *Riemann solvers and numerical methods for fluid dynamics: a practical introduction,* (2nd ed.), Springer, Berlin, 1999.

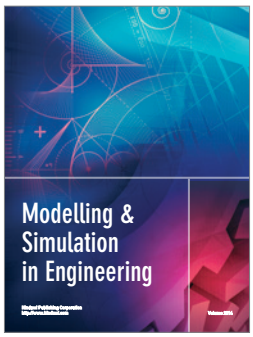


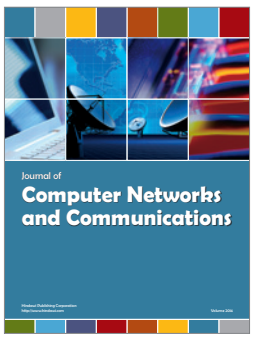












Hindawi
Submit your manuscripts at
http://www.hindawi.com